Relazione Progetto

“Inscript”

Gianluca Consoli, Michele Andrenacci,
Giovanni Babbi, Riccardo Gardenghi

25 giugno 2022

Indice

Capitolo 1

Analisi

1.1 Requisiti

Inscypt vuole essere un gioco di carte basato su alcuni giochi di carte già esistenti, quali magic, yu-gi-oh, e soprattutto Inscription. L'applicazione dovrà gestire una partita tra il Giocatore (player) e l'Avversario controllato dal computer (IA);
le 3 fasi di gioco: fase di pescata (draw phase), fase principale (main phase), fase di combattimento (battle phase);
in più la gestione del mana (serve a giocare le carte) e della vita dei giocatori;
dovrà gestire anche le carte in possesso dei due giocatori, ovvero, le carte nel mazzo, le carte nella mano e sul terreno e i loro eventuali effetti;

Le fasi:

- La fase di pescata (draw phase), nella fase di pescata, ovvero ad ogni inizio turno dei giocatori verrà aggiunta una carta dal mazzo alla mano ciò darà effettivamente inizio al turno del giocatore che passerà subito alla fase successiva.
- La fase principale (main phase), nella fase principale che si svolge subito dopo la fase di pescata il giocatore di turno potrà giocare le carte nella sua mano (se ha abbastanza mana).
- La fase di battaglia (battle phase), nella fase di battaglia le carte del giocatore di turno attaccheranno le carte che hanno di fronte.
- La fase di fine turno (end phase), la fase di fine turno avviene subito dopo la fase di battaglia e cambia il giocatore che è di turno e che dovrà ripetere tutte le fasi di gioco.

Termini il quale significato non è immediatamente intuibile:

- Gli effetti delle carte:
    1. Armored: Questo effetto si attiva solo se la carta in campo sta venendo attaccata e in quel caso il danno che subisce diventa la metà, quindi perde metà dell' attacco della carta avversaria.

2. Draw: Questo effetto si attiva quando la carta viene giocata, e fa pescare una carta dal mazzo.
3. Elusive: Questo effetto si attiva quando la carta attacca, a meno che non ci sia un'altra carta con lo stesso effetto di fronte a questa carta, la carta ignorerà la carta di fronte e attaccherà direttamente l'avversario.
4. Exalted: Questo effetto si attiva quando la carta viene giocata, le carte alla destra e alla sinistra della carta posizionata guadagnano +1 attacco.
5. Growth: Questo effetto si attiva dopo 1 turno che la carta è in gioco, la carta in questione si trasforma in un altra carta generalmente più forte.
6. Healer: Questo effetto si attiva dopo la battaglia, le carte alla destra e alla sinistra della carta.
7. Last Will: Questo effetto si attiva dopo che la vita della carta scende a 0, la carta si trasforma in un altra carta.
8. Poison: Questo effetto si attiva durante l'attacco, la vita della carta davanti a questa diventa 0.
9. Rotten: Questo effetto si attiva dopo che la vita della carta scende a 0, la carta si trasforma in un'altra carta specifica chiamata "Putridume".

- Mana: il mana è la risorsa che permette a un giocatore di giocare le proprie carte, ogni carta ha un costo di mana che varia da 1 a 10, il mana dei singoli giocatori aumenta da 1 a 10 gradualmente, ogni inizio turno il giocatore che guadagna 1 mana e recupera tutto il mana speso.

Requisiti non Funzionali:

- La View: la parte grafica che mostra la board, e i dettagli delle carte e la mano del giocatore;
- Le SoundTrack: ogni suono o canzone presente nell applicazione;
- Json: legge e crea le carte e mazzi da un json in modo da permettere una piu veloce creazione di mazzi, ma l' applicazione poteva generare autonomamente sia carte che mazzi.

1.2 Analisi e modello del dominio

l’Inscript è un'applicazione suddivisa in due principali compiti:
La selezione dei mazzi per la partita, e La sezione che permette di giocare la partita.
il gioco ha le seguenti regole:

- lo scopo del gioco è portare la vita dell’avversario a -10
- la vita di ogni giocatore parte da 0.
- se si infligge danno all'avversario si aggiunge il valore di quel danno alla propria vita
- ogni giocatore ha un quantitativo di mana da spendere ogni turno
- al primo turno il “mana massimo” del giocatore sarà 1, ogni turno tale valore andrà ad incrementare di 1 fino a raggiungere il valore di 10.
- ogni turno il valore del “mana attuale” del giocatore diventerà uguale all’ attuale valore del “mana massimo”
- i giocatori partono il gioco con 4 carte in mano e “mana massimo” settato a 1
- il campo di gioco è formato da 5 celle per le carte del giocatore e 5 per quelle dell’IA
- ogni carta ha un costo in mana per essere piazzata
- ogni carta ha un valore di attacco
- ogni carta ha un valore di vita
- ogni carta potrebbe avere un effetto
- una cella può essere occupata solo da una carta
- ogni turno il giocatore può piazzare tutte le carte che vuole dalla propria mano purché abbia celle libere e abbastanza mana per piazzarle
- una carta non può essere rimossa una volta piazzata sul campo
- se una carta giunge a 0 di vita viene rimossa dal gioco
- quando il giocatore passa il turno le carte che ha nel suo campo attaccano le carte dell'avversario
- ogni carta attacca solo la carta che si trova esattamente nella cella (nel campo dell'avversario) davanti alla propria
- se una carta attacca una cella vuota infligge danno all’avversario
- la pescata viene fatta ad inizio turno, e si pesca un'unica carta
- se un giocatore non può pescare perchè ha finito le carte il gioco finisce in parità

la partita in particolare è suddivisa in altri 3 compiti principali che corrispondono alle 3 fasi di gioco:

- la prima fase dell’applicazione detta “DrawPhase” essa dovrà gestire la pescata e il riempimento del mana del giocatore
  e l’aumento del mana massimo
- la seconda fase dell’applicazione detta “MainPhase” essa dovrà gestire la possibilità per l’utente e per AI di piazzare le proprie carte.
- la terza fase detta “BattlePhase” deve permettere alle carte di attaccare le celle dell’avversario, ed il giocatore avversario, non che distruggere le sue carte

la difficoltà primaria è connettere in maniera efficace queste 3 macro sezioni di gioco, renderle abbastanza indipendenti a livello logico per poter essere usate sia dall'interfaccia grafica che dall'IA, e permettere l'attivazione di effetti durante le varie fasi di gioco

Fatta questa premessa il dominio dell'applicazione è così composto (vedi figura 1.1):

- Player: Questa entità racchiude le informazioni di un giocatore durante la partita. Dunque da essa si potrà ricavare oltre che la vita e il mana del giocatore, anche le carte che ha in mano, sul terreno e nel mazzo
- Card: Questa entità racchiude le informazioni di una carta. in particolare la vità, l'attacco, il costo in mana e l'effetto
- Effect: Questa entità contiene le informazioni sull'effetto, ma soprattutto espone un metodo per usare l'effetto
- ActivationEvenet: è un enumeratore che identifica quando un effetto dovrebbe essere attivato
- GameMaster: Questa entità si occupa di mettere insieme i Player e i PhaseManager, e gestisce lo scorrimento del gioco.
  Questa interfaccia ha subito poi una forte modifica (vedi figura 1.2) le cui motivazioni sono spiegate meglio nella parte architetturale
- DrawPhaseManager: entità che esporrà i metodi necessari ed eseguire la drawPhase
- MainPhaseManager: entità che esporrà i metodi necessari ed eseguire la MainPhase
- BattlePhaseManager: entità che esporrà i metodi necessari ed eseguire la BattlePhase

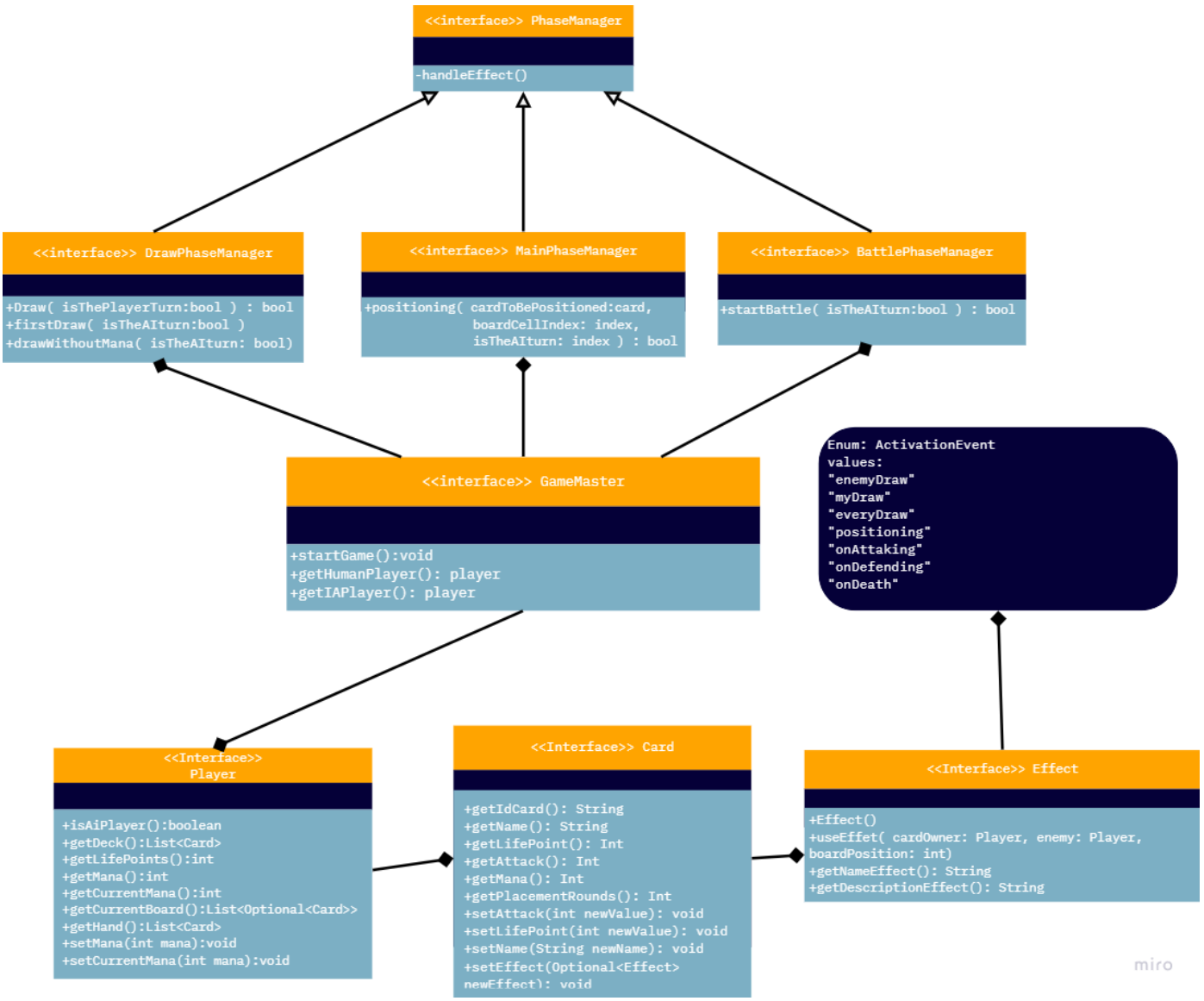


figura 1.1 - mostra le interfacce principali di Inscript. Qui non è presente l'interfaccia AppState o l'interfaccia MainPhaseManagerAI che sono state definite in seguito

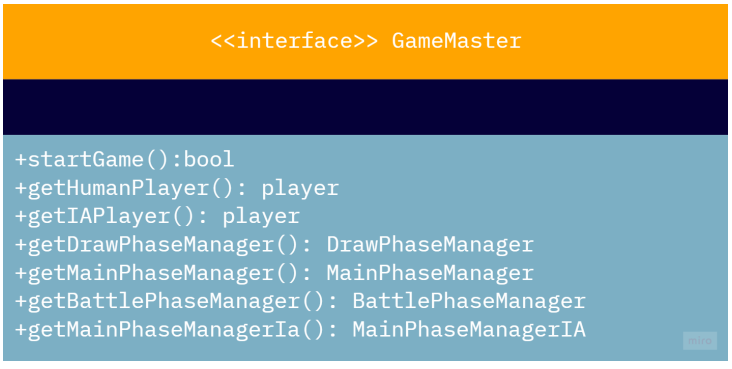


figura 1.2 - Interfaccia gamemaster dopo la fine degli sviluppi

Capitolo 2

Design

2.1 Architettura

L’Architettura di Inscript è stata fatta seguendo il modello MCV (vedi figura 2.1). per tale ragione l’applicazione è sostanzialmente divisa nelle tre macro parti. Come spiegato nell'analisi, l’applicazione deve sostanzialmente svolgere due principali compiti, la selezione dei 2 mazzi per la partita e lo svolgimento di essa.
Pertanto all'avvio l'applicazione mostrerà un'interfaccia grafica che permetta all'utente di selezionare il mazzo. Tale interfaccia è gestita a livello View dal *SelectionSceneController* esso a sua volta chiamerà la classe *AppStateContoller* che gli esporrà i metodi per modificare l’ *AppStateSingleton* ovvero la classe del Model che contiene le informazioni sui deck disponibili e sui due deck selezionati per la partita.
Invece lo svolgimento della partita viene gestito dall *GameMasterController* esso passa al *GameSceneController* i metodi necessari alle interazioni con la grafica e le informazioni sulla partita che devono essere visualizzate.
A sua volta il *GameMasterController* riceve dal *GameMaster* i metodi logici per eseguire le singole fasi della partita oltre a ricevere a sua volta le informazioni aggiornate sui Player dopo che è stata eseguita una Phase.

L’applicazione del MVC è stata tale da permettere un notevole isolamento tra View e Controller se la view cambiasse anche radicalmente il controller non ne risentirebbe.
L’unica violazione del MVC è dovuta al campo “ImageUrl” informazione relativa al path dell'immagine che è associata ad una carta. Tale campo non vincola in maniera indissolubile View e Controller. Però chiaramente resta un informazione prettamente grafica nel Modello. La soluzione a tale problema sarebbe stato creare una classe che assegnasse le immagini alle carte lato view ma per motivi di tempo non siamo riusciti ad implementarla.

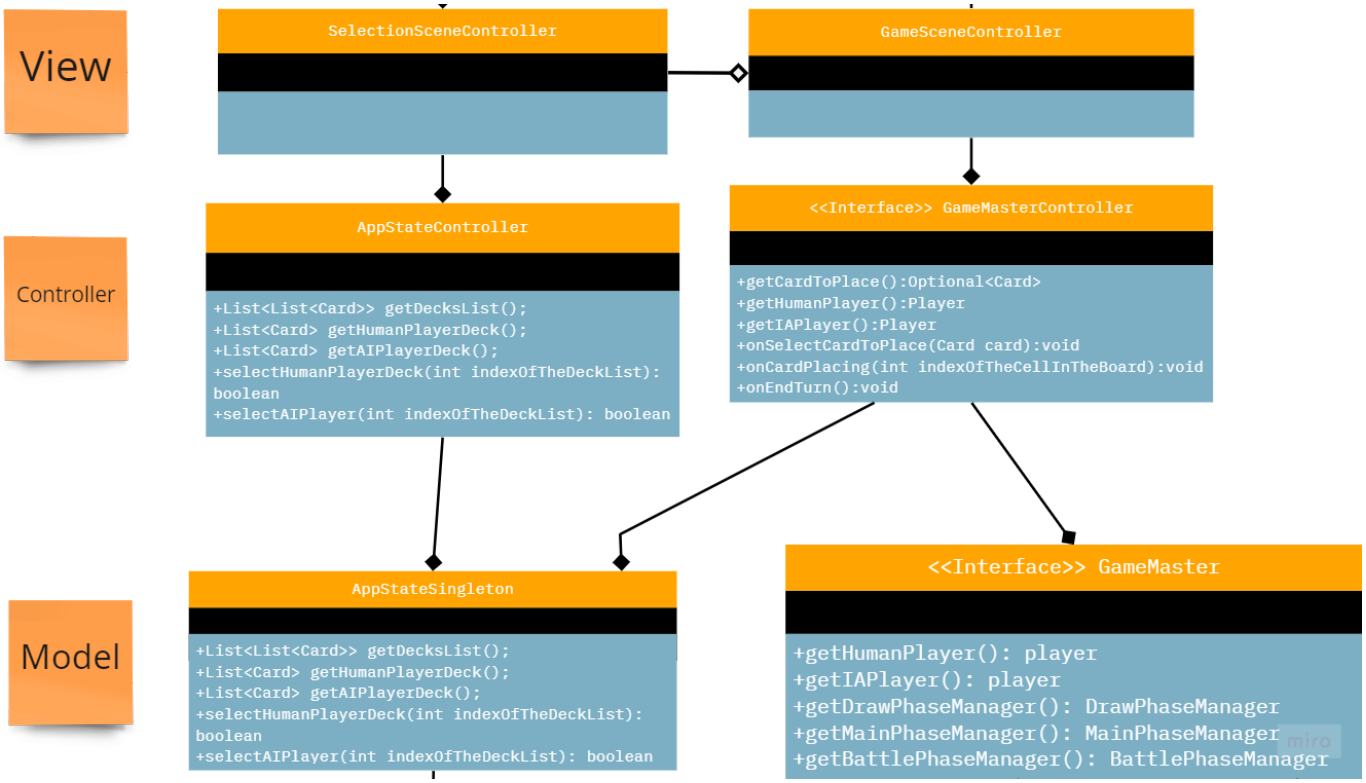


figura 2.1 - mostra la parte centrale dell' applicazione dove la view il controller ed il modello si incontrano

Scendendo più nel dettaglio del modello. La lista dei mazzi salvati nel AppStateSingleton Vengono generati a partire da un file Json (vedi figura 2.2).
Per quanto Concerne invece gli effetti delle carte essi vengono invocati dai PhaseManager che controllano per ogni carta se il suo effetto dovrebbe essere attivato in quella Phase.

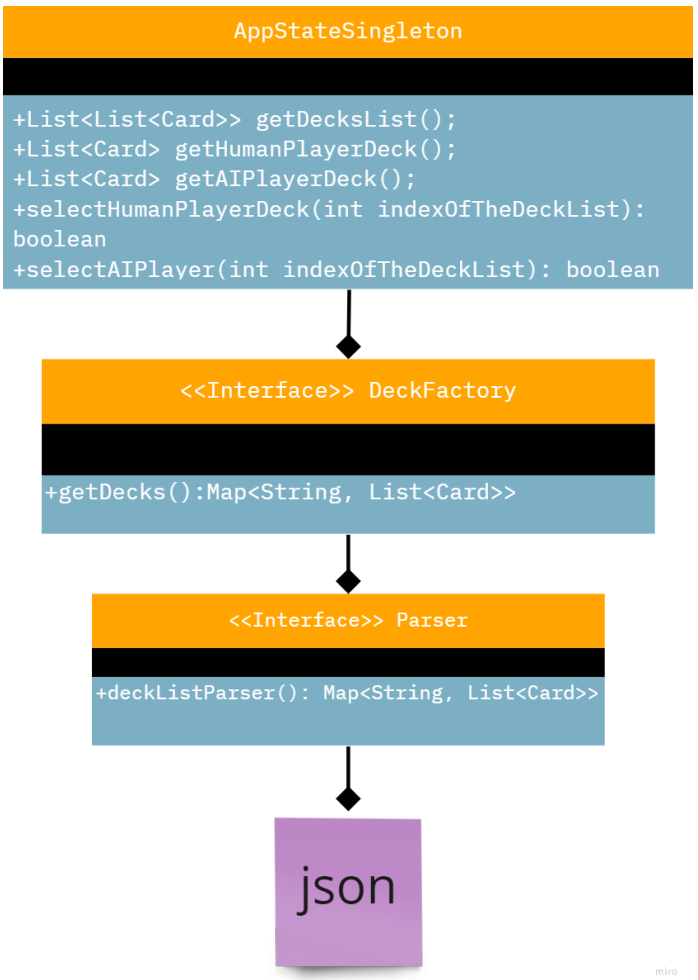


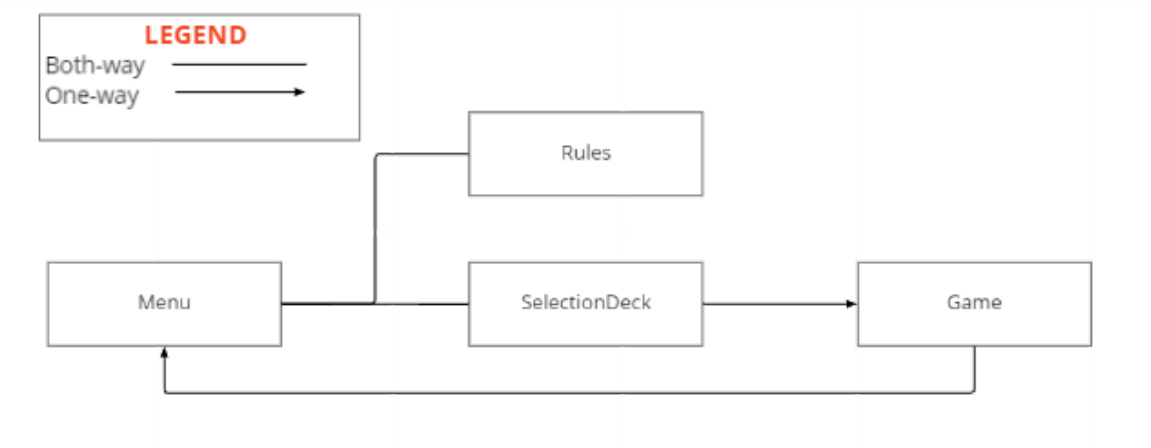
figura 2.2 - mostra in sintesi la struttura che popola la decksList dello stato

2.2 Design dettagliato

Gianluca Consoli

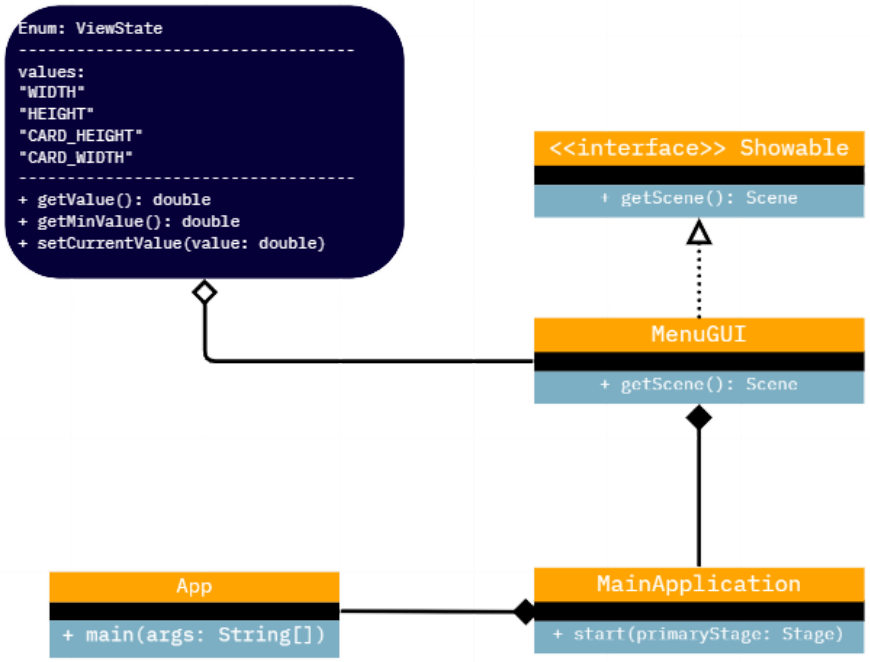
View

La parte grafica ricade nella parte della view, dove sono state utilizzate 4 scene in cui l'utente può navigare e si può orientare attraverso questo schema:



La grafica che abbiamo usato fa parte della libreria esterna di JavaFX, la decisione che ci ha portato ad utilizzare questa libreria sono state la sua flessibilità e il suo look-and-feel personalizzabile rispetto a Swing (Java). In ogni componente della grafica viene implementata l'interfaccia **Showable** in cui viene restituita la sua scena, attuando questa scelta progettuale abbiamo risolto il problema su un eventuale aggiunta di una nuova componente grafica dell'applicazione.

Figura 2.3



Nella figura 2.3 viene rappresentato il punto di partenza in cui inizia la nostra applicazione.

In quanto abbiamo usato JavaFX tutto parte dalla classe **MainApplication** che a sua volta farà partire la scena principale cioè il Menù. Per far sì che ogni scena avesse la stessa dimensione è stata creata l'enumerazione **ViewState** (usata sia per le dimensioni delle finestre dell'applicazione che delle carte del gioco). Condividendo tra di loro le stesse dimensione si può notare che passando da una scena ad un'altra non avviene in automatico ridimensionamento della finestra. Infine, sempre per quanto riguarda quest'ultima enumerazione è stato usato un modo per ri adattare le dimensioni della finestra per ogni tipo di schermo del computer.

Controller per un file FXML

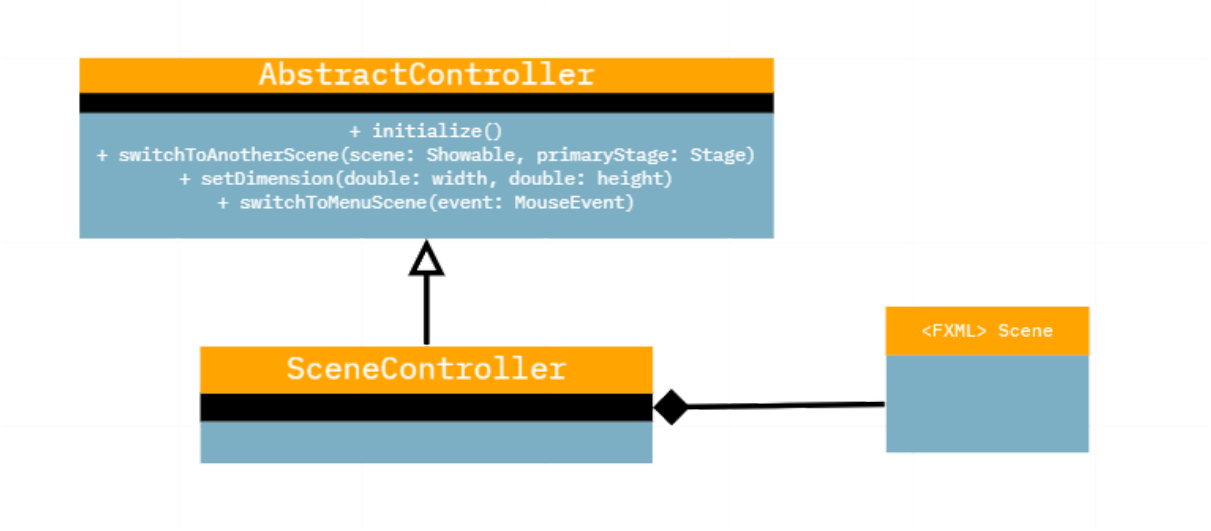


Figura 2.4: Rappresentazione UML del pattern *Template Method* per la gestione dei controllers sui file FXML

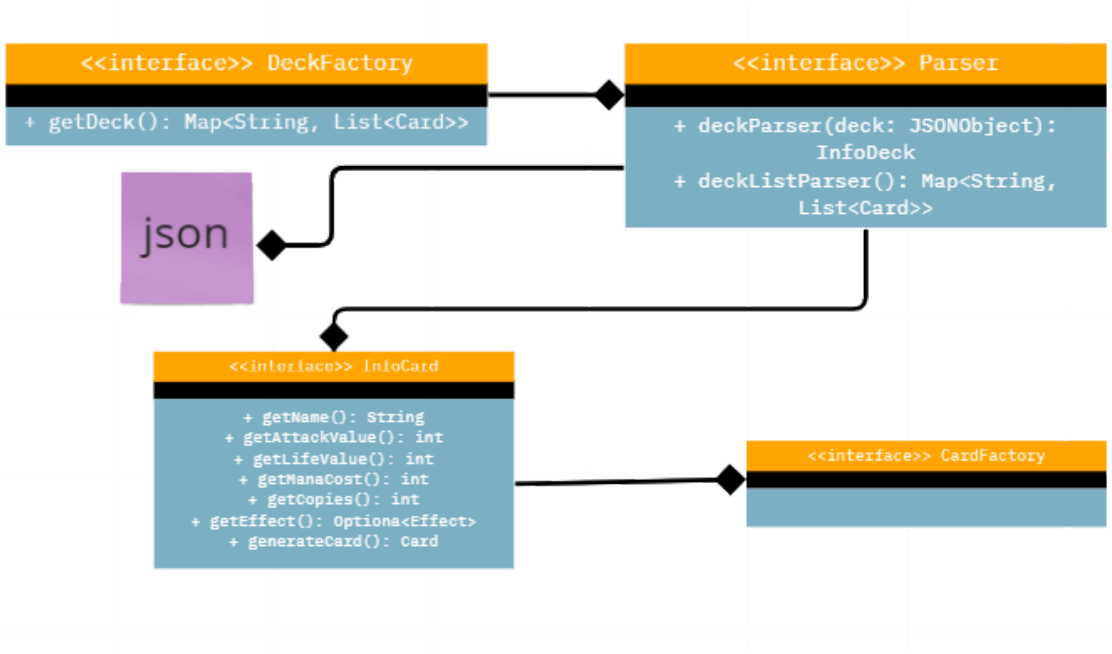
Problema: Inscript è composto da numerosi file .fxml a cui ad ognuno di essi deve essere associato ad un controller.

Soluzione: Per la gestione di un controller per ogni FXML è stato utilizzato il pattern *Template Method*: come da figura 2.4: in questo modo ogni controller ha la possibilità di inizializzare ciò che gli serve per mettere in relazione View e Controller dell'applicazione.

All'interno di ogni controller viene gestita anche la navigazione tra le scene del gioco, gestendo in questa maniera i controller il principio *SRP* non viene rispettato in quanto ogni controller ha più responsabilità, una soluzione sarebbe stata quella di creare un unico controller che gestiva la navigazione di tutta l'applicazione, ma non è stato possibile farlo proprio perché un file .fxml può essere associato ad un singolo controller e non a multipli controllers.

Json

Figura 2.5

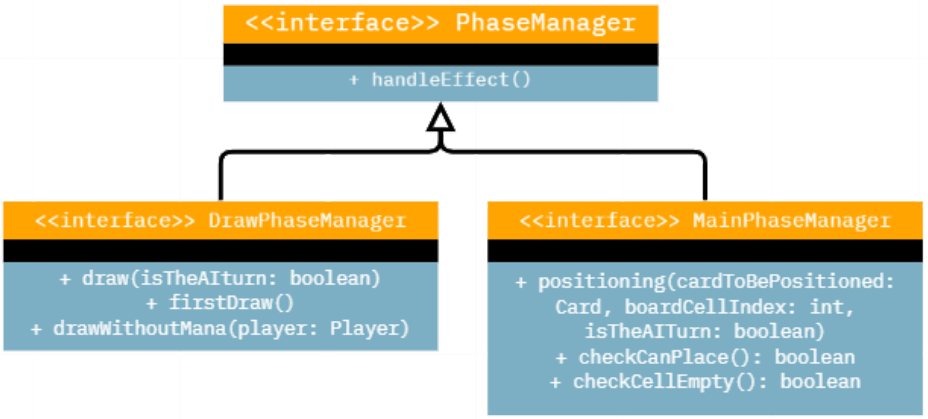


Nella Figura 2.5 viene rappresentata la struttura UML del parsing fatto sul file .json.

L'architettura sul parsing del JSON è stata fatta in collaborazione con Giovanni Babbi. Per la creazione dei deck viene usata l'interfaccia **DeckFactory** che richiama il **Parser**, a questo punto il Parser provvederà a raccogliere tutte le informazione prese dal JSON per poi creare il Deck e le Carte usabili in gioco da ogni giocatore.
InfoCard è una interfaccia che inoltre a restituire le informazioni della singola carta derivanti dal parsing, provvederà a generare la carta grazie all'interfaccia **CardFactory**.

Model - DrawPhaseManager & MainPhaseManager

Figura 2.5.2



Nella Figura 2.5.2 viene rappresentata la struttura UML dei PhaseManager (parzialmente)

Problema: All'interno del gioco quando uno dei due giocatori passa il turno all'altro avvengono tre fasi principali: *DrawPhase*, *MainPhase*, *BattlePhase*, bisognava farsì che in ogni phase avvenisse l'aggiornamento dello stato di una carta o più, in base all'effetto che aveva.

Soluzione: Ogni fase internamente, implementa il proprio modo per l'attivazione del effetto di ogni carta sul campo in base alla fase in cui si è.

La parte logica della *DrawPhaseManager* si occupa della pescata iniziale (in cui entrambi i giocatore pescano 4 carte),della pescata generale e del ristoro del mana di uno dei due giocatori; mentre, la parte logica del *MainPhaseManager* si occupa del posizionamento di una carta sul tavolo (prendendo la carta dalla mano) e del controllo di due fattori nel momento del piazzamento:

- se il giocatore poteva piazzare la carta in base a quanto mana aveva;
- se il punto in cui voleva piazzarla non era già occupata da un'altra carta.

Questi controlli dovevano essere esposti anche per la parte grafica, in quanto venivano mostrati a quest'ultima, per mancanza di tempo non è stata implementata una finestra o dialog che doveva apparire all'utente, per fargli capire che non era possibile piazzare una carta sul tavolo per le due condizioni che sono state elencate prima.

Riccardo Gardenghi

View - SceneController

Come spiegato precedentemente per la grafica è stata usata la libreria JavaFX, della quale abbiamo usato la possibilità di creare FXML a cui associare dei controller.
Inscript possiede ben 4 controller che sono:
RuleSceneController,*MenuSceneController*,*SelectionSceneController*
e *GameSceneController*.
i controller rispettano tutti lo schema mostrato in figura 2.4, ma a differenza dei primi due che occupano compiti prettamente grafici, il *SelectionSceneController* ed il *GameSceneController* si occupano di interagire con il layer di controntrollo (vedi figura 2.1).

SelectionSceneController

problema permettere all'utente di selezionare i mazzi per la partita in una pagina diversa da quella in cui viene giocata la partita
soluzione la soluzione presa è stata quella di salvare i deck scelti in un singleton (vedi figura 2.6). Singleton è stato usato siccome lo stato dell'applicazione è univoco e non si dovrebbe permettere all'app di avere più coppie di mazzi selezionate in diverse parti dell'app. Pertanto il *GameSceneController* sarà in grado di accedere ai mazzi selezionati dal *AppStateController* semplicemente accedendo allo stesso singleton. Un altra soluzione a questo problema potrebbe essere quella di usare una navigazione parametrica nella view che passasse i dati sui deck selezionati al *GameSceneController,* anche se questa soluzione non rispetterebbe propriamente MVC. Altrimenti un'altra possibile soluzione potrebbe essere una ristrutturazione della View che porti questi due controller ad essere uno solo.
Questa soluzione sarebbe agevolmente applicabile anche in vista di alcuni sviluppi futuri quali: permettere la partita fra due Giocatori in locale, oppure l'aggiunta di un altra pagina per la personalizzazione dei mazzi.

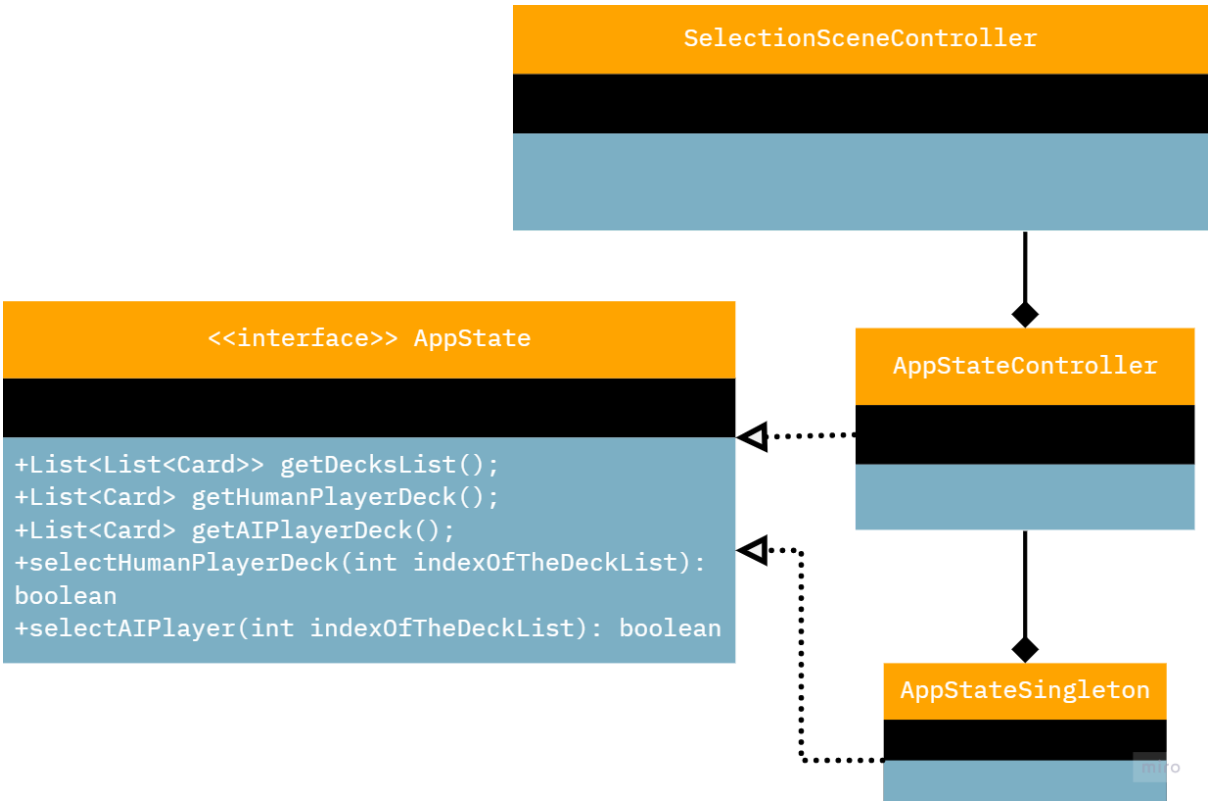


figure 2.6 - collegamento tra il SelectionScene il controller (AppStateController) ed il modello (AppStateSingleton)

GameSceneController

problema far aggiornare la grafica del campo di gioco in relazione alle operazioni svolte dall'utente senza rompere il modello MVC
soluzione pattern Strategy. Sono state dichiarate due interfacce funzionali *UpdateView* e OnGameEnd. La prima implementa il metodo che deve essere invocato quando la grafica della partita deve venir aggiornata a seguito di una giocata, La seconda implementa un metodo che verrà chiamato quando la partita è giunta al termine.
Come si può vedere in figura 2.7 la classe GameScene controller implementa al suo interno le due interfacce funzionali, inoltre come mostrato anche in figura 2.8 al suo interno la classe inizializza un'istanza dei *GameMasterControllerImpl* tale classe necessita le due interfacce funzionali come parametri del suo costruttore.

Molti elementi grafici sono stati generati in maniera dinamica dal *GameSceneController*. Per evitare che diventasse una "God class" alcune parti grafiche sono state delegate alle classi *BalanceOfLifeGraficImpl* e *CardGraficImpl*

questa soluzione è in grado di gestire cambiamenti ed evoluzioni. se si volessero ad esempio implementare altre porzioni grafiche non si dovrebbe nemmeno coinvolgere il controller. invece nel caso si voglia implementare qualche meccanica logica and esempio messaggi d'errore oppure riepiloghi delle mosse delle carte basterebbe scrivere un'altra inferfaccia funzionale e passarla al controller.

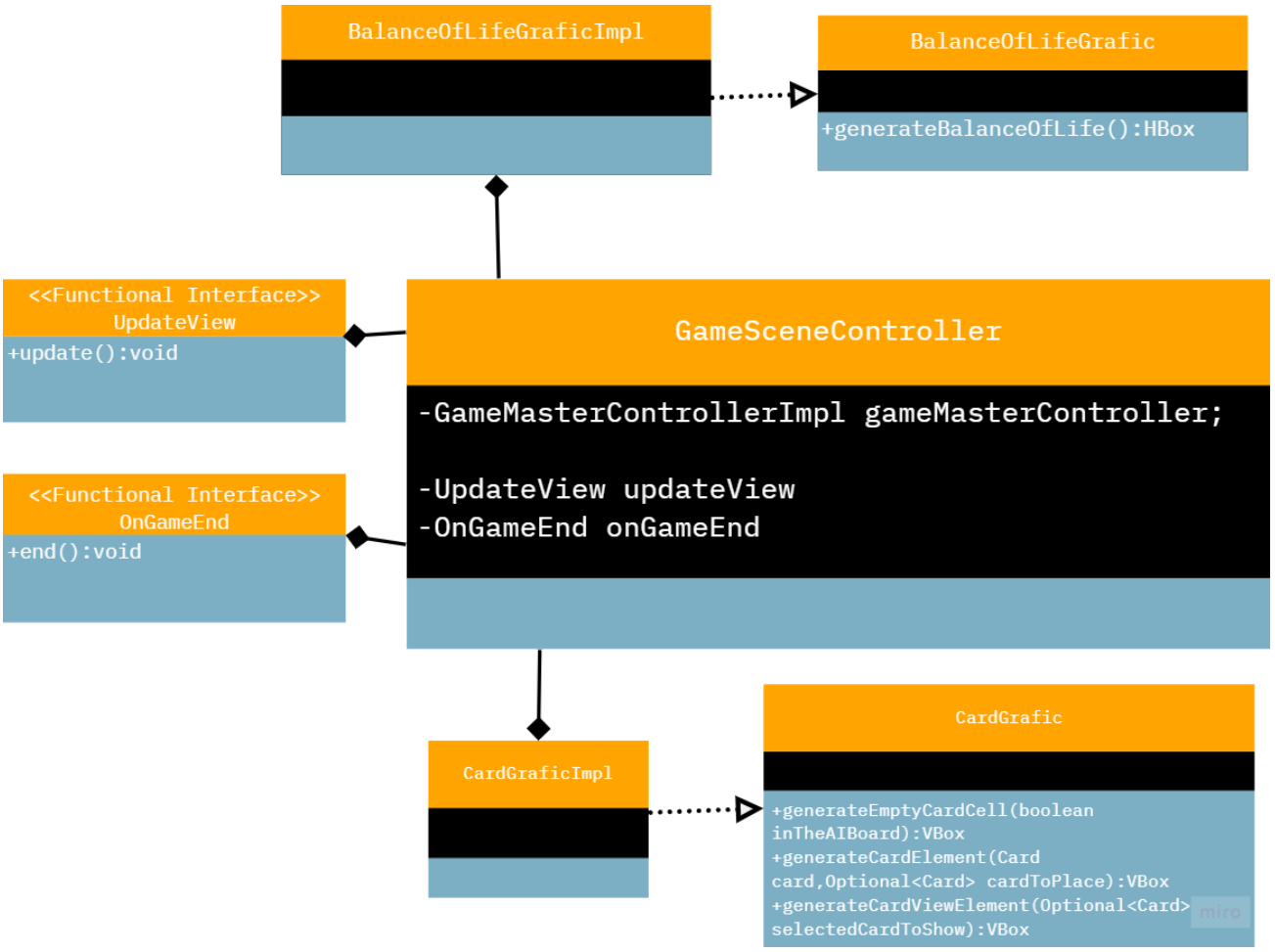


figura 2.7 mostra le altre classi grafiche collegate al *GameSceneController*. da questo schema sono esclusi l'fxml GameScene e il GameMasterControllerImpl

Controller - GameMasterController

Problema: Iniziare una partita, permettere lo scorrimento delle fasi di gioco. gestire l'aggiornamento della grafica, gestire il fine partita.
Soluzione: Come accennato precedentemente il *GameMasterControllerImpl* viene inizializzato all'interno del *GameSceneController* esso seguendo il pattern Strategy passa nel costruttore i due metodi *UpdateView* e *OnGameEnd*. A sua volta appena inizializzato il *GameMasterControllerImpl* andrà ad inizializzare un'istanza di *GameMasterImpl* alla quale passerà i due mazzi selezionati salvati nel *AppStateSingleton* (figura 2.8).
A questo punto il *GameMasterControllerImpl* ha tutto ciò che gli serve per esporre i metodi per la grafica e l'interazione con essa. In particolare il metodo *onEndTurn* che esegue in successione tutte le fasi al passaggio del turno fino a riportare l'utente alla sua MainPhase.
Dopo ogni operazione eseguita il *GameMasterControllerImpl* aggiorna la grafica con la funzione *UpdateView* e controlla se il gioco è finito ed eventualmente esegue *OnGameEnd*

La soluzione applicata è indipendente rispetto a modifiche interne al modello o alla view. la classe è però più fragile nel caso si voglia cambiare il flusso funzionale della grafica, ad esempio permettendo all'utente di vedere gli aggiornamenti causati da ogni singola fase con possibili interazioni. in questo caso la funzione *onEndTurn* andrebbe spezzata in più metodi

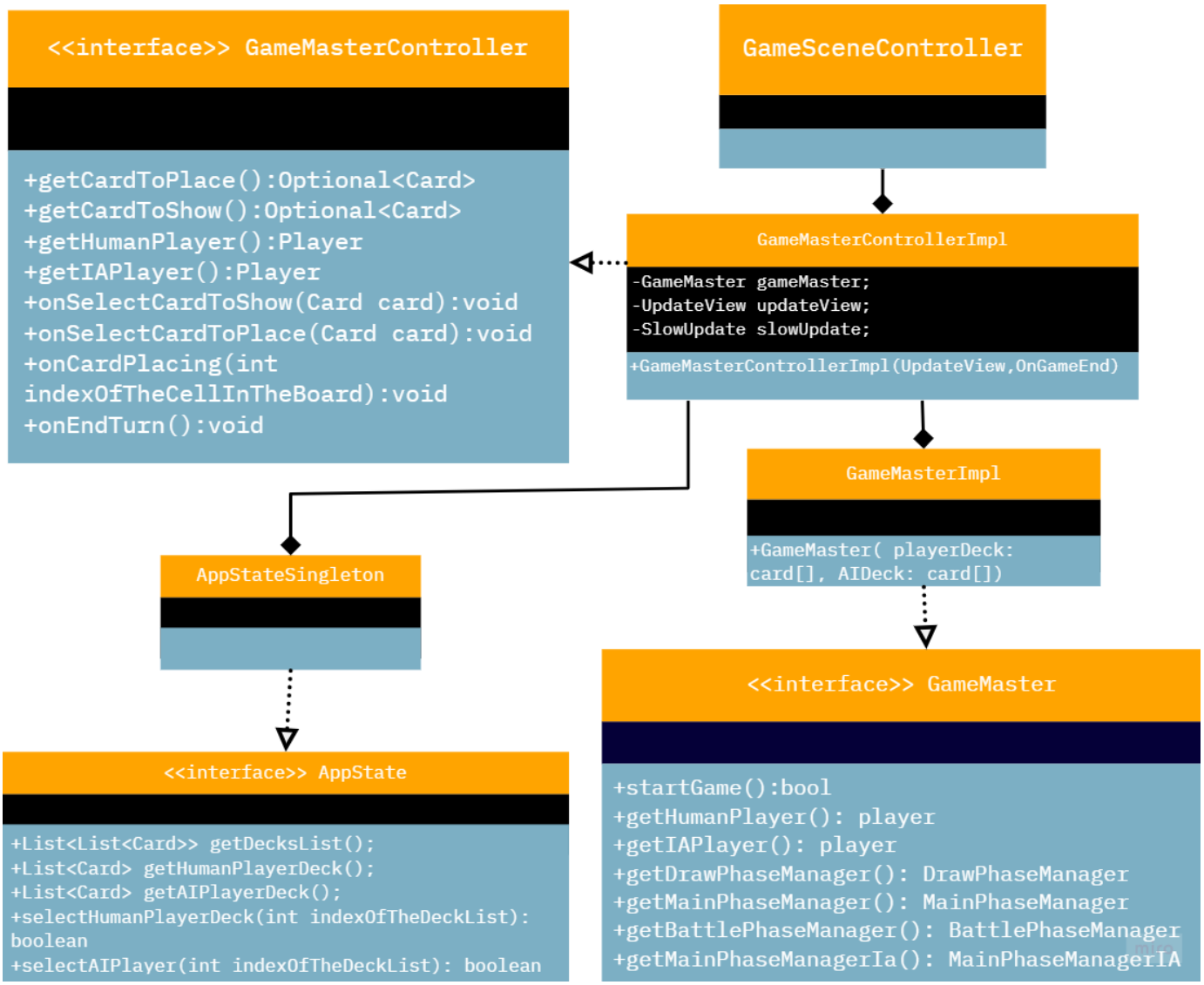


figura 2.8 - la figura mostra le classi e interfacce coinvolte nell'inizializzazione del *GameMasterControllerImpl*

Model - GameMaster

Problema: Fare in modo che gli effetti Aggiornino lo stato della partita. Esporre metodi che permettano di eseguire le fasi. inizializzare uno Stato della partita
Soluzione: quando la classe *GameMasterImpl* viene inizializzata nel suo costruttore essa inizializza a sua volta le classi dei *PhaseManager* e due Istanze di *Player*. Le due Istanze di *Player* di fatto contengono tutte le informazioni della partita (figura 2.9). Queste due istanze vengono passate dall *GameMasterImpl* ai *PhaseManager* come argomenti dei loro costruttori.
Il *GameMasterImpl* espone tutte queste Istanze le quali saranno poi usate dal *GameMasterController* ed il *GameSceneController*.
Gli effetti delle carte vengono eseguiti nel momento opportuno dai PhaseManager. Siccome ai PhaseManager vengono passati i Player gli effetti sostanzialmente sono in grado di fare qualsiasi tipo di modifica alla partita.

Questo problema come i *PhaseManager* e il *GameMasterControllerImpl* (precedentemente citato) potevano essere sviluppati con un po più di coerenza applicando il pattern Template method sui PhaseManager. Inoltre si sarebbe dovuto applicare il Pattern Observer per la gestione Degli effetti, facendo in modo che gli effetti delle carte stessero in ascolto degli eventi trasmessi dalle fasi nel momento in cui eseguivano le loro operazioni o dal esecuzione degli effetti di altre carte

Questa architettura è sicuramente in grado di resistere agevolmente a cambi logici all'interno dei singoli *PhaseManager*, se però si volesse aggiungere un altra fase di gioco le interfacce andrebbero cambiate. Invece non è necessario nessun Cambiamento alla creazione di nuovi effetti.
Inoltre un grande debolezza di questa architettura è insita nel fatto che gli effetti vengono eseguiti dalle singoli *PhaseManager*, perciò non è possibile creare un effetto che agisca in risposta all'attivazione di un altro effetto. Ma solo effetti che Interagiscono in relazione a ciò che c' è nello stato corrente della partita nel momento dell'attivazione.

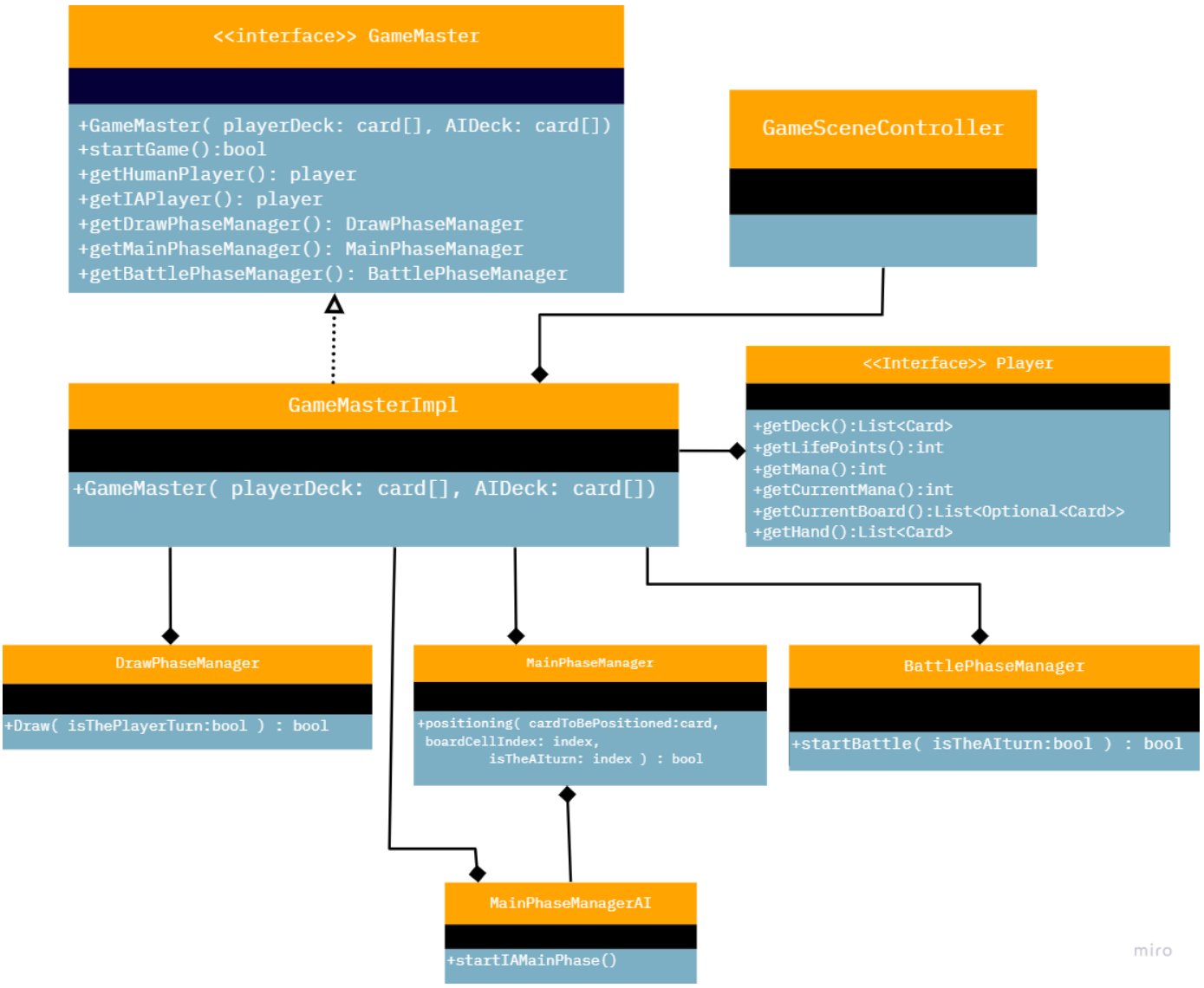


figura 2.9 - Mostra le principali classi coinvolte nel inizializzazione gameMaster, non sono mostrate le interfacce dei PhaseManager

Model - MainPhaseManagerIA

Problema: gestire la mainPhase del IA, ovvero l'IA osservando il cambo scegli dalla sua mano le carte che vuole piazzare e le posiziona in campo
Soluzione: come si può vedere nella figura 2.10 il *MainPhaseManagerIAimpl* utilizza al suo interno il *MainPhaseManagerImpl* del quale viene inizializzata un'istanza. Di fatto la classe espone un unico metodo, ovvero *startAiMainPhase* esso sceglie guardando la propia mano quale è la carta piu forte che puo essere piazzata e la usa per distruggere la carta che ritiene piu forte tra quelle nel campo avversario.
Questa classe è flessibile rispetto hai cambiamenti sia del *GameMasterImpl* che del *MainPhaseManagerImpl* siccome espone un solo metodo e non piazza le carte in maniera autonoma ma si affida ad l'univoca gestione di piazzamento delle carte stabilita dal *MainPhaseManagerImpl.* Anche se la gestione degli effetti venisse aggiornata ad usare il pattern Observer questa classe non ne risentirebbe.

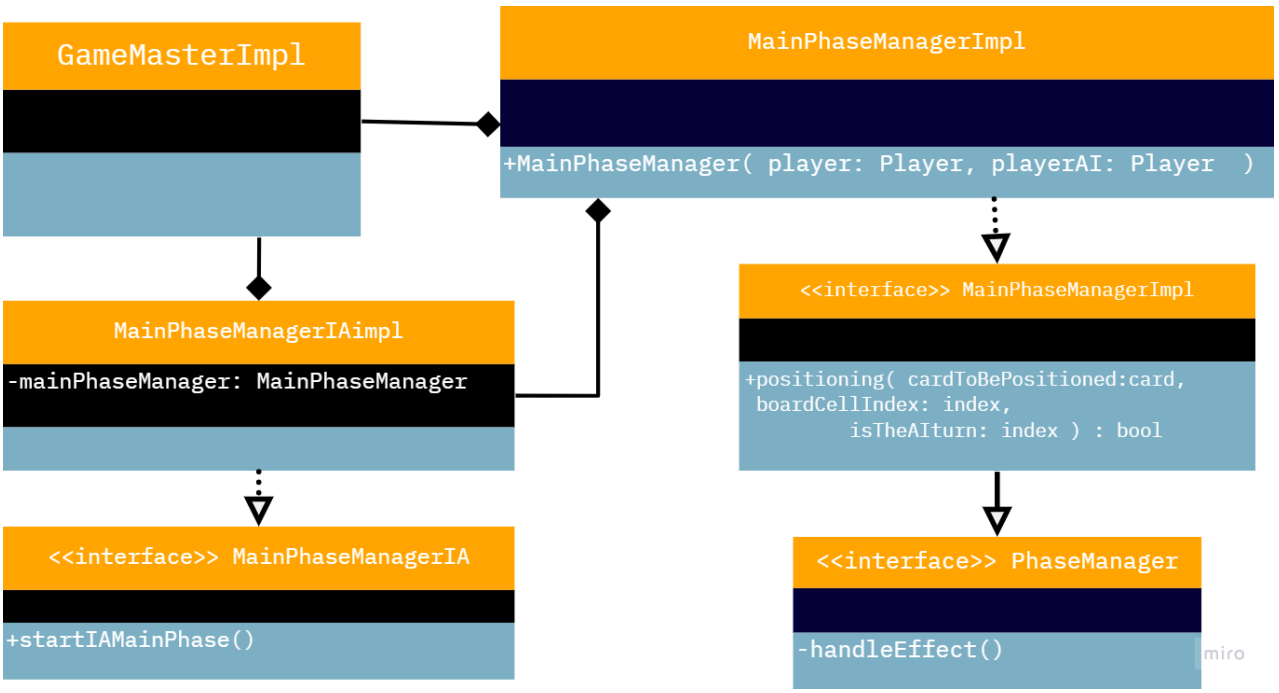


figura 2.10 - mostra i PhaseManager che si occupano della MainPhase del AI e del giocatore

Model - CardIDgenerator

Problema: assegnare ID univoci ad ogni carta
Soluzione: si è scelto di usare il pattern singleton (figura 2.10.2). Questa scelta è dovuta al fatto che gli ID devono essere generati in maniera univoca e non ci possono essere piu possibili fonti di generazione degli id che causerebbero discordanza

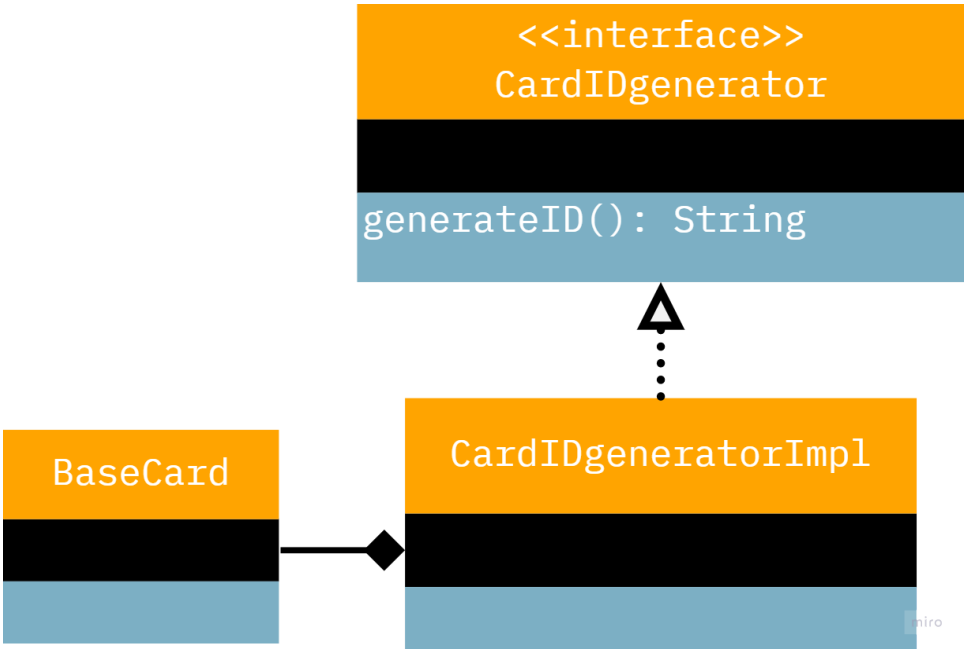


figura 2.10.2 - mostra i come è la struttura che generà l'id della carta

Andrenacci Michele

Model - Card and Effect

Le interfacce Card e Effect sono la parte centrale di questa sezione dell'UML poiché collegate a tutte le altre classi o interfacce, per una migliore leggibilità è stato diviso in tre parti più semplici.

Per prima cosa andava creata una struttura base che andasse bene per ogni carta perciò creando un'interfaccia Card contenente tutti i metodi utili per la singola carta e poi implementando la classe BaseCard come effettiva Classe di riferimento per le carte (figura 2.11) abbiamo creato una solida base di inizio per la creazione delle carte. Questa base ci ha poi portato alla costruzione degli effetti che sono stati implementati successivamente.
Ogni carta ha come parametro:
idCard, che serve a distinguere le carte uguali,
name, attack, lifeValue e mana sono le caratteristiche principali della singola carta,
placementAround serve a contare i turni nella quale la carta è sul terreno di gioco,
effect è l'attributo più importante, visto che definisce il comportamento della carta.

Gli Effetti sono Optional<Effect> poiché potrebbero esistere carte che non hanno alcun effetto, come si può dedurre da figura 2.13 per gli effetti è stato usato il Pattern Decorator, tramite la classe astratta AbstractEffect si definiscono i campi e i metodi che gli effetti dovranno avere, per poi fare un override del singolo effetto in base alla funzione che devono svolgere. [N.B. per rendere l'UML più leggibile e meno confusionario nella figura 2.13 non sono stati inseriti tutti gli effetti esistenti ma solo i 2 "tipi" principali; effetti che non richiedono parametri come ad esempio Poison, ed effetti che come LastWill chiedo dei parametri poiché mutano la carta stessa].
Usiamo un enumerazione, ActivationEvent, per definire quando un effetto deve attivarsi.

Nell'immagine 2.12 sono state utilizzate due factory per rendere più agile e autonoma la creazione di singole carte e mazzi. La factory che crea i mazzi utilizza la factory che crea le carte, richiamando l'effetto che la carta deve avere e aggiungendola automaticamente nel mazzo.
La creazione delle carte è stata implementata dopo l'implementazione di tutti gli effetti, la CardFactory è molto semplice, i suoi metodi sono chiamati come i nomi degli effetti presenti nel gioco, creando così una carta con quell effetto, ciò implica che se si vuole aggiungere un effetto bisogna necessariamente aggiungere un metodo al CardFactory. Gli effetti sono quasi tutti simili tra loro l'unica differenza, che non ha influito sul pattern decorator è la presenza di alcuni effetti come per esempio LastWill che richiede dei parametri aggiuntivi.
La DeckFactory richiama i metodi della CardFactory creando così delle carte e le aggiunge ad un mazzo.
I metodi CardFactory e DeckFactory fanno da tramite al json per la effettiva creazione delle carte.


BaseCard
+idCard: String
+name: String
+lifeValue: int
+attackValue: int
+manaCost: int
+effect: Optional<Effect>
+placementsRounds: int
+imageURL: String
+BaseCard(name: String, lifeValue: int, attackValue: int, manaCost: int, imageURL: String, effect: Effect)
<<Interface>> Card
+getIdCard(): String
+getName(): String
+getLifePoint(): Int
+getAttack(): Int
+getMana(): Int
+getPlacementRounds(): Int
+setAttack(newValue: int): void
+setLifePoint(newValue: int): void
+setName(newName: String): void
+setEffect(newEffect: Optional<Effect>): void
+getImageURL(): String
+setImageURL(newImageURL: String): void
+setPlacementRounds(newValue: int): void
+getEffect(): Optional<Effect>
Enum: ActivationEvent
values:
"enemyDraw"
"myDraw"
"everyDraw"
"positioning"
"onAttaking"
"onDefending"
"onDeath"
<<Interface>> Effect
+useEffect(cardOwner: Player, enemy: Player, boardPosition: int)
+getNameEffect(): String
+getDescriptionEffect(): String
+getImageEffectURL(): String
+setImageEffectURL(newImageEffectURL: String): void
+getActivationEvent(): ActivationEvent
miro

figura 2.11 -


BaseCard
+idCard: String
+name: String
+lifeValue: int
+attackValue: int
+manaCost: int
+effect: Optional<Effect>
+placementsRounds: int
+imageURL: String
+BaseCard(name: String, lifeValue: int, attackValue: int, manaCost: int, imageURL: String, effect: Effect)
<<Interface>> CardFactory
+noEffect(name: String, lifeValue: int, attackValue: int, manaCost: int, imageURL: String): Card
+elusiveEffect(name: String, lifeValue: int, attackValue: int, manaCost: int, imageURL: String): Card
+exaltedEffect(name: String, lifeValue: int, attackValue: int, manaCost: int, imageURL: String): Card
+healerEffect(name: String, lifeValue: int, attackValue: int, manaCost: int, imageURL: String): Card
+poisonEffect(name: String, lifeValue: int, attackValue: int, manaCost: int, imageURL: String): Card
+armoredEffect(name: String, lifeValue: int, attackValue: int, manaCost: int, imageURL: String): Card
+rottenEffect(name: String, lifeValue: int, attackValue: int, manaCost: int, imageURL: String): Card
+drawEffect(name: String, lifeValue: int, attackValue: int, manaCost: int, imageURL: String): Card
+growthEffect(name: String, lifeValue: int, attackValue: int, manaCost: int, imageURL: String, growthName: String, growthLife: int, growthAttack: int, growthEffect: Optional<Effect>, growthImageURL: String): Card
+lastWillEffect(name: String, lifeValue: int, attackValue: int, manaCost: int, imageURL: String, lastWillhName: String, lastWillLife: int, lastWillAttack: int, lastWillEffect: Optional<Effect>, lastWillImageURL: String): Card
<<Interface>> Card
+getIdCard(): String
+getName(): String
+getLifePoint(): Int
+getAttack(): Int
+getMana(): Int
+getPlacementRounds(): Int
+setAttack(newValue: int): void
+setLifePoint(newValue: int): void
+setName(newName: String): void
+setEffect(newEffect: Optional<Effect>): void
+getImageURL(): String
+setImageURL(newImageURL: String): void
+setPlacementRounds(newValue: int): void
+getEffect(): Optional<Effect>
DeckFactoryImpl
+playerDeck: List<Card>
+playerIADeck: List<Card>
+deckList: Map<String, List<Card>>
+getPlayerDeck(): List<Card>
+getPlayerAIDeck(): List<Card>
CardFactoryImpl
<<Interface>> DeckFactory
+getDeck(): Map<String, List<Card>>
miro

figura 2.12 -

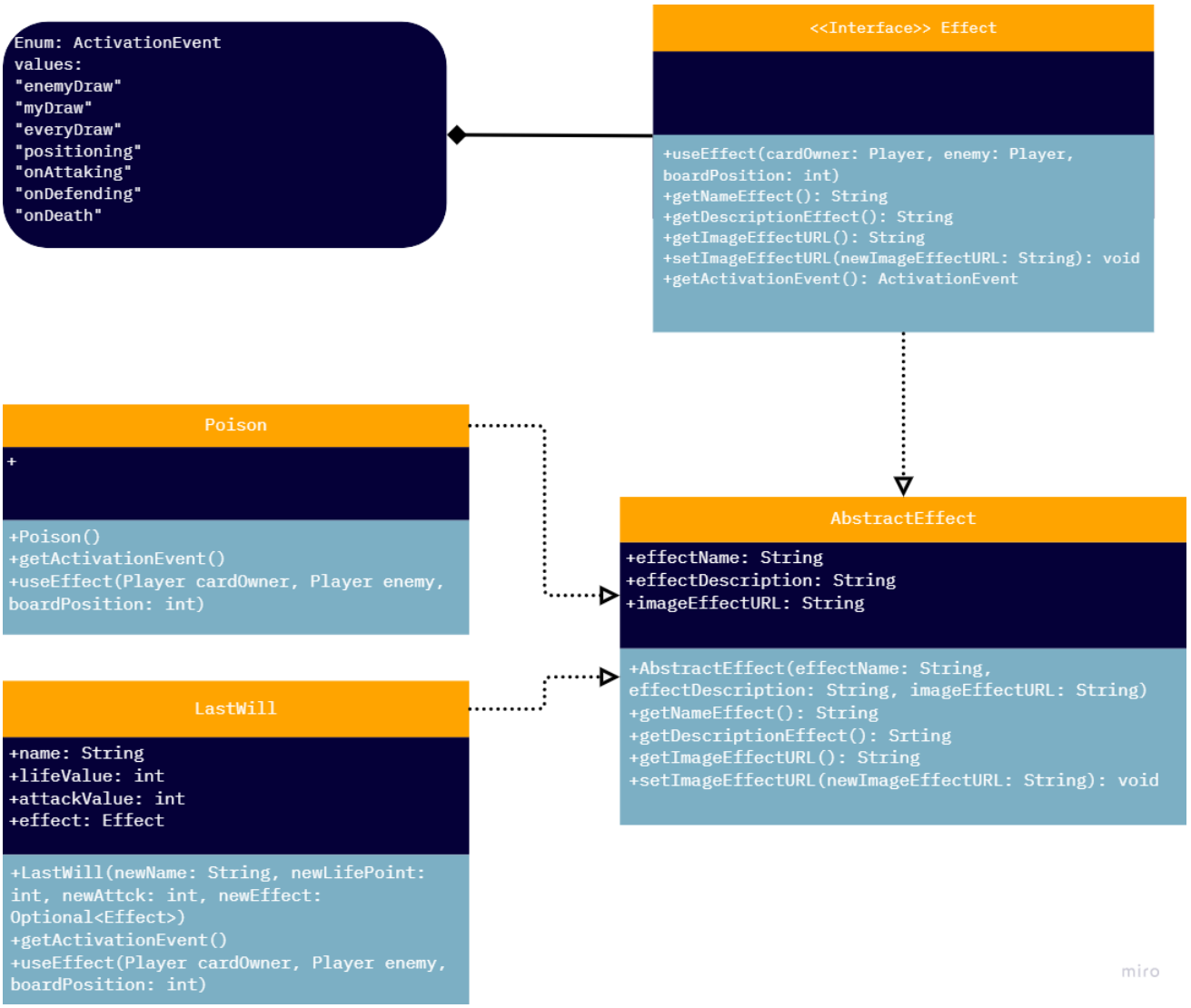

Enum: ActivationEvent
values:
"enemyDraw"
"myDraw"
"everyDraw"
"positioning"
"onAttaking"
"onDefending"
"onDeath"
<<Interface>> Effect
+useEffect(cardOwner: Player, enemy: Player, boardPosition: int)
+getNameEffect(): String
+getDescriptionEffect(): String
+getImageEffectURL(): String
+setImageEffectURL(newImageEffectURL: String): void
+getActivationEvent(): ActivationEvent
Poison
+
+Poison()
+getActivationEvent()
+useEffect(Player cardOwner, Player enemy, boardPosition: int)
AbstractEffect
+effectName: String
+effectDescription: String
+imageEffectURL: String
+AbstractEffect(effectName: String, effectDescription: String, imageEffectURL: String)
+getNameEffect(): String
+getDescriptionEffect(): Srting
+getImageEffectURL(): String
+setImageEffectURL(newImageEffectURL: String): void
LastWill
+name: String
+lifeValue: int
+attackValue: int
+effect: Effect
+LastWill(newName: String, newLifePoint: int, newAttck: int, newEffect: Optional<Effect>)
+getActivationEvent()
+useEffect(Player cardOwner, Player enemy, boardPosition: int)
miro

Figura 2.13 -

Giovanni Babbi
Model - BattlePhaseManager

Problema: Creare un componente in grado di tenere traccia dell'andamento della battaglia, con eventuali modifiche dei campi di gioco dei giocatori e gestire la corretta attivazione degli effetti durante di essa.

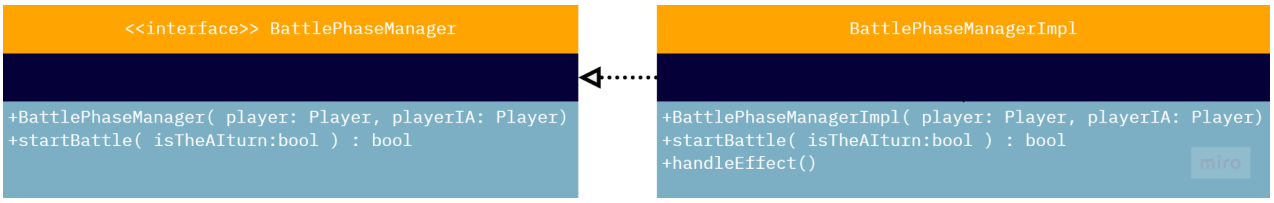


Figura 2.14 -

Soluzione: BattlePhaseManagerImpl è un classe che sfruttando dei controlli posizionali, va a simulare carta per carta, la fase di battaglia fra giocatori. Un oggetto di questa classe mette a disposizione il suo metodo principale necessario per l'inizio della gestione della battaglia, startBattle. Da cui inizierà a reperire dal campo dei giocatori i vari effetti che devono essere gestiti durante il corso della battaglia. Durante il suo svolgimento il fattore principale è se la battaglia che si sta svolgendo è quella durante il turno del giocatore o meno. Questo fattore va a definire il giocatore che dovrà difendersi e quale attaccare. Tramite questo presupposto ho determinato che chi si difendeva era l'unico che, al termine della battaglia, avrebbe notato la modifica più sostanziale del suo campo di battaglia. Per ogni carta di cui simulavo la battaglia, prima che questa infliggesse un danno alla rispettiva carta del giocatore avversario, venivano attivati gli effetti di quelle carte che dovevano essere gestiti durante le fasi attacco o difesa. Dopodiché veniva registrato il danno alla carta e, se questa moriva in battaglia, venivano gestiti anche gli effetti che si attivano alla morte. In caso chi si doveva difendere da un attacco non aveva una carta schierata in una determinata posizione, il danno veniva inflitto alla vita del rispettivo giocatore.

Il vantaggio di usare un controllo posizionale permette di controllare carta per carta l'andamento della battaglia, semplificando le interazioni con gli elementi e l'implementazione dei controlli. Inoltre, anche con una ipotetica variazione della dimensione del campo di battaglia di entrambi i giocatori(settato a 5 carte per entrambi i campi), l'efficacia del controllo persiste. Lo svantaggio principale è che rischia di rendere le fasi di controllo più complesse di quel che in realtà non sono e di rendere il codice intuitivamente meno leggibile, ciò è causato dalla presenza di svariati campi Optional che vengono gestiti al momento del loro utilizzo. In compenso, anche se si va a creare di fatto un god method, ciò permette di gestire correttamente tutti gli eventi che si verificano durante il corso della battaglia al momento opportuno.
Questa implementazione presenta leggeri accenni a svariati design pattern, ma non ne segue nessuno e, se anche fosse, sarebbe stata una cosa non intenzionale.

Model - JSON

Problema: Permettere di editare e aggiungere mazzi mediante un metodo che non implicasse la modifica diretta del codice sorgente.

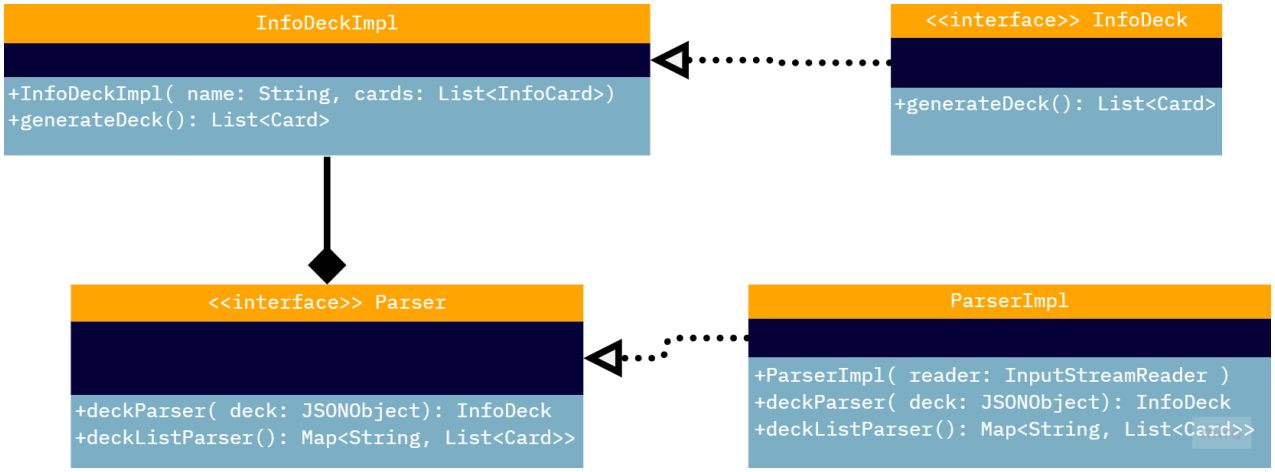


Figura 2.15 -

Soluzione: Tramite un file JSON e una struttura definita per il suddetto, l'utente è in grado di modificare e/o aggiungere deck alla lista di quelli prestabiliti dall'applicazione, a patto che mantenga la struttura utilizzata nel file JSON.
La MainApplication va ad utilizzare un oggetto di tipo ParserImpl che:
- legge il contenuto del file;
- esegue il parse del contenuto del file letto, seguendone la struttura definita;
- crea una lista di prototipi di carte con i dati ottenuti, che saranno la base per creare le carte di ogni deck.

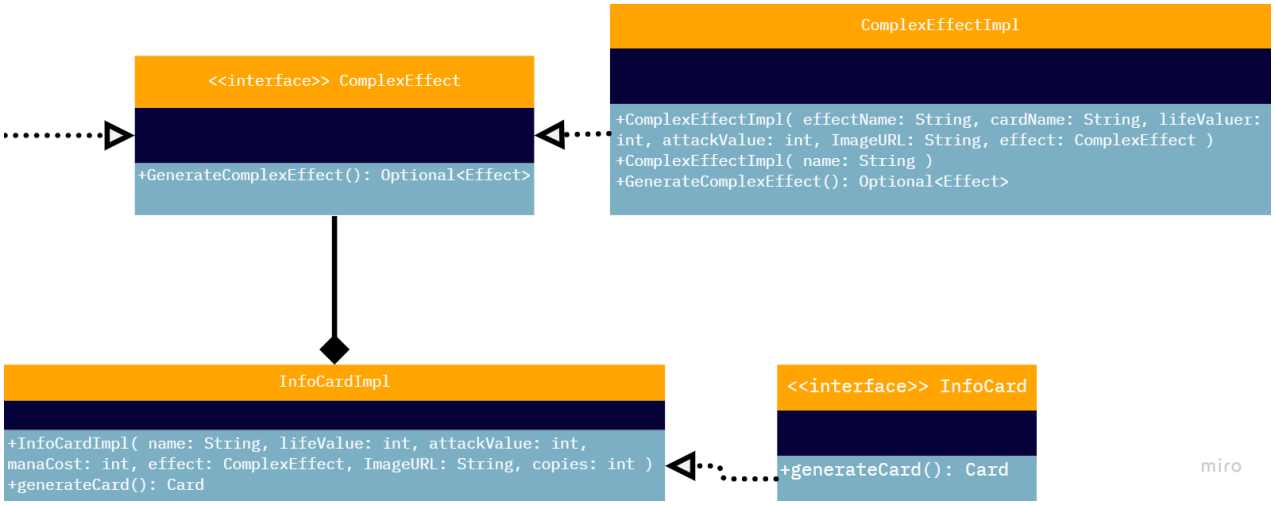


Figura 2.16 -

La creazione delle carte viene gestita tramite delle chiamate a metodo e cicli for standard, mentre per la gestione degli effetti, per mantenere l'architettura dei metodi delle classi specializzati nella creazione degli effetti delle carte, viene fatta in maniera ricorsiva.In quanto ogni carta a sua volta può essere in grado di trasformarsi in un'altra che si può ri-trasformare.

Inoltre, sempre per la gestione degli effetti ho deciso di definire dentro all'implementazione dell'interfaccia ComplexEffect un secondo costruttore che viene richiamato in caso l'effetto che deve essere creato è semplice.

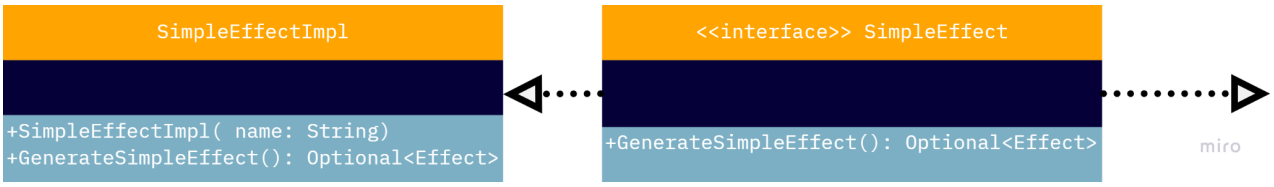


Figura 2.19 -

Con semplice si intende un effetto che si limita ad avere effetti aggiuntivi durante il corso della battaglia o del turno del giocatore.

Questo mi ha permesso di poter gestire separatamente sia dentro le classi interessate, ma anche nei loro metodi, come si dovevano comportare i relativi oggetti in caso si fossero trovati in una situazione o nell'altra.
Questa implementazione della soluzione è stata scelta in quanto permette di dividere in maniera molto efficiente i compiti dei vari componenti, ma allo stesso tempo di farli interagire in maniera adeguata. Lo svantaggio di questa implementazione è che, in caso non venga rispettata la struttura del JSON, durante la fase di parsing, ci si accorge dell'errore e viene generata un'eccezione al momento della scelta dei deck. Invece se i dati inseriti nel file sono inadeguati o incoerenti è garantita solo la corretta generazione degli effetti. Inoltre questa implementazione non usufruisce di nessun design pattern da me conosciuto.

Capitolo 3

Sviluppo

3.1 Testing automatizzato

Per testare la nostra applicazione abbiamo usato dei test automatizzati che comprendevano tutte le fasi di gioco. Per la loro esecuzione è stato usato il task “test” messo a disposizione da Gradle. In ogni test veniva visto se le diverse fasi del gioco effettivamente dovevano funzionare come avevamo posto durante l’analisi del problema (e.g: controllo della dimensione del mazzo dopo una pescata, verifica dell’assenza della carta dopo un piazzamento, …). In seguito sono stati fatti test manuali per quanto riguarda il funzionamento dell’applicazione su altri sistemi operativi (Windows, Linux, …).

Abbiamo testato con test automatici le seguenti feature:

- Effetti
- DrawPhase
- MainPhase
- BattlePhase
- il GameMasterController
- AppStateSingleton (ovvero la selezione dei mazzi)

l’app è comunque stata testata manualmente per verificare il corretto funzionamento del flusso e delle funzionalità nel loro insieme in fase di integrazione e merge tra i brench. I bug che abbiamo trovato sono stati tracciati su ClickUp (figura 3.1)
link a click up - https://app.clickup.com/31075255/v/li/187082540

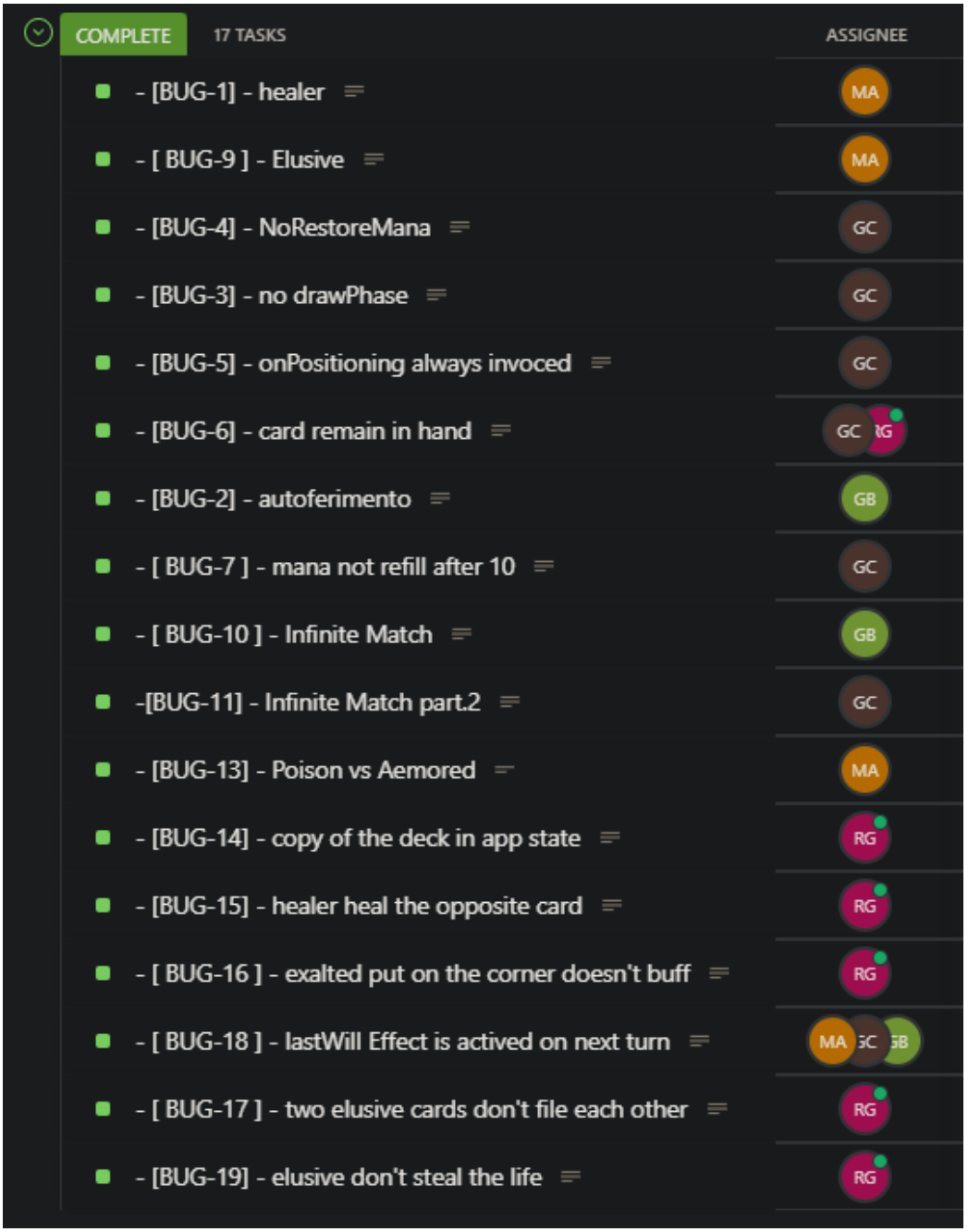


figura 3.1 - mostra i bug tracciati su ClickUp

3.2 Metodologia di lavoro

Come base di partenza abbiamo stabilito in comune accordo di seguire il pattern MVC cercando di creare meno dipendenze possibili tra loro, però se uno o più componenti del progetto avevano intenzione di fare una modifica che andava a toccare l'MVC, ci consultavamo tutti assieme e decidevamo se era una buona cosa farlo o meno. Come workflow abbiamo usato GitFlow che ci ha aiutato ad organizzare ed ad integrare le parti lavorative di ogni componente del progetto. In generale i branch erano suddivisi in questa maniera:

- master - branch in cui quando ci sono state abbastanza modifiche sul branch develop viene pushata la nuova versione del gioco
- develop - è il branch di integrazione tra i branch comuni
- feature / nome feature - sono i branch individuali in cui vengono sviluppate le feature del gioco, ogni volta che viene conclusa una feature dovrà essere mergiata su develop ed eliminata

Quando ogni componente commitava una propria modifica su un certo branch, abbiamo stabilito delle regole : definire dentro le quadre il nome del task e dopo il trattino scrivere una mini-descrizione su cosa si è fatto in quel commit.
es

```
- [ TEST ] - add test GameMasterController
```

Per ogni branch feature sono stati assegnati dei task, per task si intende l'implementazione specifica di un feature, questo è stato fatto perché c'erano alcuni branch feature che erano composti da più task.
Elenco Task:

- Shared - feature per le implementazioni di parti condivise, e setup
- Cards - feature che comporta lo sviluppo delle carte
- BoardGUI - feature che comporta la creazione della GUI della table
- RuleGUI - feature che comporta la creazione della GUI delle rules
- SelectionGUI - feature che comporta la creazione della GUI sulle selection
- CardGUI - feature che comporta la creazione della grafica delle carte
- DrawPhase - feature per lo sviluppo della DrawPhase
- MainPhase - feature per lo sviluppo della MainPhase
- BattlePhase - feature per lo sviluppo della BarrlePhase
- JSON - feature per lo sviluppo del parsing del JSON
- TEST - feature per la creazione dei test

Durante lo sviluppo delle varie feature su branch separati, abbiamo seguito questa regola:

- le interfacce comuni vanno aggiornate nel branch di integrazione (Ovvero Develop)
- tutti devono tenere i loro branch allineati a develop
- le restanti parti della propria feature devono venir sviluppate esclusivamente nel branch di competenza

seguire questi principi ci ha permesso di evitare merge-conflit di tipo architetturale. Inoltre se veniva modificata un'interfaccia comune di solito esso veniva notificato nel nostro gruppo di lavoro e segnato nel UML del progetto permettendo a tutti di esserne consapevoli in tempo reale.

Durante la fase di integrazione siccome era difficile capire in quale feature risiedeva ogni abbiamo deciso di tracciare i bug e le irregolartà trovate su ClickUp in modo che non venissero dimenticate (figura 3.1).
La descrizione del bug su ClickUp Doveva inoltre includere:

- step necessari per riprodurre tale bug;
- il comportamento previsto di quella sezione di codice afflitta da bug;
- il comportamento attuale del codice congestionato dal bug; (modificato)

Inoltre quando uno si prendeva in carico un bug su ClickUp andava ad assegnarselo e a settare il suo stato come "*in progress*" e una volta finito lo segnava come "*completed*" ed il fix andava commitato seguendo questo pattern di commit

- [NomeDellaFeatureInteressataDalBug-NumeroDelBug] - TitoloRiassuntivoDelBug

Divisione Del lavoro

Riccardo Gardenghi:

- Definizione Iniziale Interfacce Del Dominio (Model)
- *GameMaster* (Model), *GameMasterController* (Controller)
- *AppStateSingleton* (Model), *AppStateController* (Controller)
- *GameSceneController* (View)
- *MainPhaseManagerIA* (Model)
- *BalanceOfLifeGrafic* (View)
- *Rule.fxml* (View), *RuleSceneController* (View)
- CardIDGenerator (Model)
- supporto

Gianluca Consoli

- Setup Progetto git
- *DrawPhaseManager* (Model)
- *MainPhaseManager* (Model)
- AbstractController (View)
- Music, ViewState (View)
- MainApplication (View)
- MenuSceneController (View)
- GameScene.fxml, MenuScene.fxml, SelectionScene.fxml (View)
- supporto

Giovanni Babbi

- Implementazione dell'interfaccia BattlePhaseManager (Model)
- Implementazione della classe BattlePhaseManagerImpl (Model)
- Implementazione del packege jsonparser(Model)
- Modifica del file di build del progetto Gradle
- Supporto

Michele Andrenacci

- Implementazione dell'interfaccia *Card* (Model)
- Implementazione della classe Card(Model)
- Implementazione dell'Interfaccia *Effetc* (Model)
- Implementazione della classe Astratta AbstractEffect (Model)
- Implementazione tramite AbstracEffect e pattern decorator di tutti gli effetti (Model)
- *CardFactory* (Model)

Giovanni Babbi e Michele Andrenacci

- *CardGrafic* (View)

Gianluca Consoli e Riccardo Gardenghi

- Setup Gradle
- GameScene.fxml (View)
- SelectSceneController (View)

Gianluca Consoli e Giovanni Babbi

- *Parser* (Model)
- Scrittura del Json (Model)

Collettivi

- Scrittura dei Test
- Implementazione dell'interfaccia Player
- Interfaccia Effect
- bug fixing integrazione
- *DeckFactory* (Model)

3.3 Note di sviluppo

Riccardo Gardenghi

- utilizzo di Optional: Gli Optional sono stati vastamente usati nell'applicazione per quanto riguarda il "campo" della partita. infatti il campo è composto da 5 celle ma non è detto che vi siano delle carte sopra. Per questo motivo sia nelle classi della grafica che nel *MainPhaseManagerIAimp*l sono stati usati per fare diversi controlli e agire di conseguenza
- utilizzo di Stream: Gli Stream sono stati utilizzati massivamente per scorrere la mano dei giocatori, il loro deck o il loro campo di gioco. ed eseguire operazioni su di essi.
- javaFx: javaFX è stato usato per produrre tutta la grafica dell'applicazione. in quanto tale, ho avuto modo di usarlo per generare parte del campo di gioco e delle carte (con lo sviluppo del *GameSceneController*), non che la grafica della bilancia della vita, la pagina delle regole ed il *SelectSceneController.*
  Qui cito l'articolo che ho seguito per cercare di sviluppare MVC con javaFX
  https://edencoding.com/mvc-in-javafx/
- Fuctional Interface e labda Expression: come anche citato nel dettaglio dell'architettura ho utilizzato le interfacce funzionali e le labda expression per implementare il pattern strategy nel *GameSceneController*
- utilizzo di Gradle

Gianluca Consoli

- utilizzo di Optional: Gli Optional sono stati utilizzati in modo ampio sia per la presenza di una carta nel tavolo che per la presenza di effetti all'interno di una carta. DrawPhaseManager, MainPhaseManager, la View e il pasing del JSON ne hanno fatto un grande utilizzo.
- utilizzo di Stream:Gli Stream sono stati utlizzati per il controllo dell'attivazione di un effetto di ogni carta, ma anche per l'estrazione di uno o più decks prima dell'inizio del gioco.
- utilizzo di annotazioni: Per l'individuazione di campi o metodi ricollegabili ai file .fxml sono state utilizzate le annotazioni @FXML, in più sono state usate quelle per i testing e quelle basi.
- CSS: Per lo stile testuale del gioco e la grafica del gioco è stato usato un file css.
- utilizzo di librerie esterne: Per la creazione di tutta la grafica dell'applicazione sono stati usati i file FXML usando JAVAFx; mentre la creazione dei decks è stata fatta tramite lettura di un file JSON.
- utilizzo di Gradle: Sono stati utilizzati dei task per creare dei fat Jar e per far eseguire i test dell'applicazione.

Andrenacci Michele

- Utilizzo di Optional: Abbiamo utilizzato molto gli Optional sia per gli effetti delle Carte sia per la composizione del campo di gioco
- Utilizzo di Pattern: Per gli effetti è stato utilizzato il pattern decorator, mentre per la creazione di carte e mazzo sono state utilizzate delle factory
- JavaFx: sviluppo della grafica del dettaglio della carta
- Utilizzo di Gradle

Giovanni Babbi

- utilizzo di Optional: Gli Optional sono stati utilizzati in maniera intensiva sia per gestire situazioni in cui non c'erano ne carte durante le fasi di battaglia ne effetti nelle rispettive carte da attivare. Sono stati utilizzati massivamente anche per la creazione dei mazzi via file JSON.
- utilizzo di Stream: Sono stati utilizzati per la parte di creazione e successiva generazione dei deck partendo dalla lista di elementi InfoList.
- Utilizzo di org.json: nello specifico org.json.simple per eseguire il parsing del file JSON e per interagire con i componenti del file. Il codice che utilizza questa libreria è stato creato usando come base i concetti e i codici presenti al seguente link: **https://howtodoinjava.com/java/library/json-simple-read-write-json-examples/**
- Utilizzo di Gradle: è stato modificato il file di building del progetto gradle per aggiungere org.json.simple.

Capitolo 4

Commenti finali

4.1 Autovalutazione e lavori futuri

Riccardo Gardenghi
Avendo preso in carico la parte centrale del progetto ho praticamente settato l'architettura del core del progetto. Nonostante a livello di integrazione tra le parti non abbiamo avuto problemi, l'architettura che ho fatto per il funzionamento del GameMaster sarebbe stata meglio se fosse stata sviluppata dal pattern Observer e purtroppo non svilupparla in quel modo ha influenzato lo sviluppo di molte feature collegate ad essa, Ovvero i PhaseManager e gli Effetti. Questo errore è stato causato da una scarsa analisi

Gianluca Consoli
A me sono state affidate due parti molto interessanti, tra cui la grafica con l'uso di JavaFX (View) e il parsing del JSON. Lavorando sulla grafica in questo progetto, mi ha aiutato a capire molto meglio come programmarla e soprattutto quali problemi possono esserci ma anche come risolverli di conseguenza. Tutto ciò mi servirà in un lavoro futuro nel caso mi chiedessero di lavorarci sopra. Mentre invece, lavorando sulla lettura di un file JSON, mi ha fatto capire come creare un oggetto tramite delle informazioni ricavabili da un singolo file, che possa essere capito da un umano; dato che è usato molto spesso anche in ambito lavorativo penso che sia stato molto utile imparare il suo funzionamento e il suo impego.

Giovanni Babbi
La mia parte di lavoro consisteva principalmente nello sviluppo di un componente in grado di gestire la fase di battaglia e di un componente in grado di creare i deck partendo dal file JSON. Successivamente mi sono state assegnate anche alcune piccole parti della sezione grafica, ma era più per sistemare parti incoerenti, oppure per assistere qualcuno nella creazione di un componente. In fin dei conti direi che il modo in cui ho sviluppato la mia parte sia abbastanza interessante, ma comunque presenta una carenza di riutilizzabilità in alcuni punti e una chiara assenza di design patterns, che potevano aiutare in tale scopo. Comunque lo reputo lo stesso un buon lavoro in quanto penso di aver usato al meglio le conoscenze di cui potevo disporre durante il periodo di sviluppo e implementazione.

4.2 Difficoltà incontrate e commenti per i docenti

Riccardo Gardenghi

ritengo che il progetto in se sia una delle cose più formative che abbia fatto all'università fino ad ora. L'unico appunto che farei è che lo sviluppo della grafica anche quello di una CLI è comunque dispendioso.
Inoltre la conoscenza di un minimo di logica grafica di come funziona la grafica specialmente di sistemi che usano linguaggi di markup è una competenza sicuramente apprezzabile. Perciò sarebbe interessante che anche il lavoro lato View venisse valorizzato nonostante il progetto sia di "Programmazione ad oggetti"

Gianluca Consoli

penso che aver fatto questo tipo di progetto ci abbia fatto capire quanto sia difficile lavorare in team (se uno non l'ha mai fatto), archiviando questa nuova esperienza quando uno lavorerà (in futuro) in uno nuovo progetto composto da più persone saprà già come comportarsi e che errori non dovrà fare. Le cose viste in classe, mi hanno dato una mano per raggiungere la cosiddetta buona programmazione e tra gli argomenti più utili, personalmente vedo: i patterns, i principi da seguire e l'UML. Le uniche difficoltà che ogni tanto ho riscontrato sono stati gli argomenti che sono stati trattati poco durante il corso tra cui: l'uso dei FXML, la creazione di uno di questi file e gradle.

Giovanni Babbi

Durante la mia parte di sviluppo una delle problematiche che ho incontrato era durante l'implementazione di una soluzione ricorsiva in un componente del parser, avendo diverse difficoltà nello sviluppo in maniera ricorsiva anche quando queste sono effettivamente più efficaci.
Un'altra problematica che ho incontrato era durante lo sviluppo del BattlePhaseManager, in cui ho cercato in più modi di implementare handleBattle in maniera differente da come è risultato alla fine, ma senza successo, in quanto gli Stream non erano in grado di gestire gli elementi di tipo Optional, e non avevo idee migliori se non quella di fare un confronto delle carte in base alla loro posizione con un banale for.

Appendice A

Guida utente

Una volta avviato il file .Jar, l'applicazione si apre con una schermata con 3 voci principali:

- **Start Game**
- **Rules**
- **Exit**

Start Game: porterà a una schermata in cui sarà possibile scegliere quale mazzo giocare e quale far giocare all'AI avversaria. Di default sono settati dei mazzi standard su cui abbiamo basato l'intero sistema di bilanciamento di tutti gli altri mazzi disponibili. Si può scegliere un deck differente facendo uso degli appositi menù a tendina per entrambi i giocatori.

Rules: mostra una schermata a scorrimento verticale che descrive, con l'ausilio di immagini, le regole del gioco, come si svolgono le fasi e cosa accade durante il corso della partita.

Exit: si limita semplicemente a terminare il programma.

Una volta scelto il deck e iniziata la partita ci si ritrova davanti alla schermata di gioco.

Questa è la board di gioco, in cui è possibile posizionare le carte presenti nella mano. La prima fila in basso rappresenta il campo di gioco del giocatore, mentre quella in alto è il campo della AI. Le carte presenti nella stessa colonna si affronteranno, infliggendo danni, nel caso non ci fosse una carta da parte del giocatore o dell'AI, l'eventuale danno sarà inflitto direttamente ai Life Points del rispettivo giocatore.

Localizzata subito sotto il campo di battaglia del giocatore si possono vedere le carte che sono presenti nella propria mano. Una partita inizia con entrambi i giocatori aventi 4 carte in mano ciascuno.
Per posizionare una carta sul campo di battaglia bisogna prima selezionarla, cliccando col tasto sinistro sulla carta che si vuole evocare e poi cliccare sulla casella del proprio campo in cui si vuole posizionare la carta scelta.

Per poter evocare una carta è necessario disporre di una certa quantità di mana.

Sulla sinistra della board di gioco è presente il mana disponibile per il turno attuale, una bilancia che tiene traccia dei Life Points di entrambi i giocatori e il pulsante per far iniziare a combattere le carte presenti sul proprio campo.

Non c'è un limite al numero di carte posizionabili sul proprio campo di battaglia, a patto che si disponga del mana necessario per poterle evocare. Però non è possibile sostituire una carta presente nel proprio campo fino a che questa non viene uccisa dall'avversario.

La partita finisce quando, dopo la fase di battaglia uno dei 2 giocatori ha subito 10 danni totali.

I danni inflitti nei vari turni persistono anche in quelli successivi.

Sulla destra del campo di battaglia invece, è presente una rappresentazione più dettagliata della carta selezionata in quel momento.

Rispetto alla rappresentazione delle carte in mano, questa ha come fattore in più la descrizione dell'effetto che la carta possiede, se questa ne ha uno.

Inoltre il pulsante per tornare al menù principale sarà localizzato in tutte le schermate in alto a destra, tranne per il menù di selezione del deck in cui è locato in basso a destra vicino al pulsante per iniziare la partita.

Nella menù iniziale in basso a destra, e nella schermata di gioco in alto a sinistra, è anche presente un pulsante per mutare la musica di sottofondo.

Si consiglia di avviare l'applicazione con un volume non troppo alto(Circa tra 20 - 50%)

Si consiglia inoltre di leggere le regole di gioco, in cui sono spiegati in modo più dettagliato le dinamiche della battaglia, gli effetti delle carte e l'avanzamento dei turni; oltre che a una spiegazione più esaustiva dei componenti, trattati nella guida, disponibili nella schermata di gioco.

Appendice B

Esercitazioni di laboratorio

B.0.1 Gianluca Consoli

- Laboratorio 06: https://github.com/Giock24/OOP-Lab06 ;
- Laboratorio 08: https://github.com/Giock24/OOP-Lab08 ;
- Laboratorio 09: https://github.com/Giock24/OOP2021-Lab09 ;
- Laboratorio 10: https://github.com/Giock24/OOP2021-Lab10 .